场景:lc1860平台 camera分配内存失败

*[ 1357.566459]{0} vmap allocation for size 113983488 failed: use vmalloc=<size> to increase size.*

*[ 1357.576274]{0} comip-camera a0500000.camera: alloc mem for isp of size 113977344 failed*

Vmalloc.c (g:\code-pj\source\lc1860\linux-3.10.y\mm) 69917 2021/12/1

static struct vmap\_area \*alloc\_vmap\_area(unsigned long size,

unsigned long align,

unsigned long vstart, unsigned long vend,

int node, gfp\_t gfp\_mask)

{

struct vmap\_area \*va;

struct rb\_node \*n;

unsigned long addr;

int purged = 0;

struct vmap\_area \*first;

BUG\_ON(!size);

BUG\_ON(size & ~PAGE\_MASK);

BUG\_ON(!is\_power\_of\_2(align));

va = kmalloc\_node(sizeof(struct vmap\_area),

gfp\_mask & GFP\_RECLAIM\_MASK, node);

if (unlikely(!va))

return ERR\_PTR(-ENOMEM);

retry:

spin\_lock(&vmap\_area\_lock);

/\*

\* Invalidate cache if we have more permissive parameters.

\* cached\_hole\_size notes the largest hole noticed \_below\_

\* the vmap\_area cached in free\_vmap\_cache: if size fits

\* into that hole, we want to scan from vstart to reuse

\* the hole instead of allocating above free\_vmap\_cache.

\* Note that \_\_free\_vmap\_area may update free\_vmap\_cache

\* without updating cached\_hole\_size or cached\_align.

\*/

if (!free\_vmap\_cache ||

size < cached\_hole\_size ||

vstart < cached\_vstart ||

align < cached\_align) {

nocache:

cached\_hole\_size = 0;

free\_vmap\_cache = NULL;

}

/\* record if we encounter less permissive parameters \*/

cached\_vstart = vstart;

cached\_align = align;

/\* find starting point for our search \*/

if (free\_vmap\_cache) {

first = rb\_entry(free\_vmap\_cache, struct vmap\_area, rb\_node);

addr = ALIGN(first->va\_end, align);

if (addr < vstart)

goto nocache;

if (addr + size < addr)

goto overflow;

} else {

addr = ALIGN(vstart, align);

if (addr + size < addr)

goto overflow;

n = vmap\_area\_root.rb\_node;

first = NULL;

while (n) {

struct vmap\_area \*tmp;

tmp = rb\_entry(n, struct vmap\_area, rb\_node);

if (tmp->va\_end >= addr) {

first = tmp;

if (tmp->va\_start <= addr)

break;

n = n->rb\_left;

} else

n = n->rb\_right;

}

if (!first)

goto found;

}

/\* from the starting point, walk areas until a suitable hole is found \*/

while (addr + size > first->va\_start && addr + size <= vend) {

if (addr + cached\_hole\_size < first->va\_start)

cached\_hole\_size = first->va\_start - addr;

addr = ALIGN(first->va\_end, align);

if (addr + size < addr)

goto overflow;

if (list\_is\_last(&first->list, &vmap\_area\_list))

goto found;

first = list\_entry(first->list.next,

struct vmap\_area, list);

}

found:

if (addr + size > vend)

goto overflow;

va->va\_start = addr;

va->va\_end = addr + size;

va->flags = 0;

\_\_insert\_vmap\_area(va);

free\_vmap\_cache = &va->rb\_node;

spin\_unlock(&vmap\_area\_lock);

BUG\_ON(va->va\_start & (align-1));

BUG\_ON(va->va\_start < vstart);

BUG\_ON(va->va\_end > vend);

return va;

overflow:

spin\_unlock(&vmap\_area\_lock);

if (!purged) {

purge\_vmap\_area\_lazy();

purged = 1;

goto retry;

}

if (printk\_ratelimit())

printk(KERN\_WARNING

"vmap allocation for size %lu failed: "

"use vmalloc=<size> to increase size.\n", size);

kfree(va);

return ERR\_PTR(-EBUSY);

}

解决方案

./include/configs/comip\_lc1860\_evb2\_w.h:235:#define CONFIG\_BOOTARGS\_CHARGE\_ONLY        "loglevel=0 console=ttyS3,115200 vmalloc=412M " CONFIG\_BOOTARGS\_RAMDISK CONFIG\_BOOTARGS\_MEM "android "

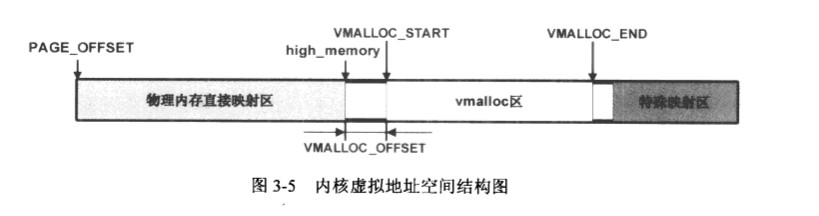
需改改

./include/configs/comip\_lc1860\_evb2\_w.h:235:#define CONFIG\_BOOTARGS\_CHARGE\_ONLY        "loglevel=0 console=ttyS3,115200 vmalloc=812M " CONFIG\_BOOTARGS\_RAMDISK CONFIG\_BOOTARGS\_MEM "android "

[linux内存管理之vmalloc](http://blog.chinaunix.net/uid-20786208-id-4888173.html)

分类： LINUX

2015-03-13 11:07:24

     在前面我们讲解了kmalloc申请连续物理内存的操作，以及原理和基础cache . 在内核中还有另外一个接口函数那就是vmalloc，申请一片连续的虚拟地址空间，但不保证物理空间连续，实际上我们会想到用户空间的malloc，malloc它是标准的glibc封装的一个函数，最终实现是通过系统调用brk和mmap来实现，以后在分析它的实现过程. 它就是申请连续的虚拟空间，但是不保证物理内存的连续，当然用户程序也不怎么关心这个问题，只所以会关心物理内存的连续性一般是由于设备驱动的使用，或者DMA.  但是vmalloc申请效率比较低，还会造成TLB抖动. 一般内核里常用kmalloc. 除非特殊需求，比如要获取大块内存时，实例就是当ko模块加载到内核运行时，即需要vmalloc.   
释放函数：vfree   
参考内核  3.8.13  
  这里是说32位的处理器，即最大寻址4G虚拟空间，（当然现在已经64位比较普及了，后续补上吧）而虚拟地址到物理地址的转化往往需要硬件的支持才能提高效率，即MMU。  
当然前提需要os先建立页表PT. 在linux内核，这4G空间并不是完全给用户空间使用在高端0xC0000000 （3G开始）留给内核空间使用（x86默认配置，默认0-16M（DMA），16M-896M（Normal），896M-1G（128M）作为高端内存分配区域），当然这个区域也是可是配置的.）.  
kmalloc函数返回的是虚拟地址(). kmalloc特殊之处在于它分配的内存是物理上连续的,这对于要进行DMA的设备十分重要. 而用vmalloc分配的内存只是连续,物理地址不一定连续,不能直接用于DMA。我们可以参考一个图：（它是arm 32架构的内核虚拟地址分配图）  
  
下面我们就看看vmalloc函数：(mm/vmalloc.c)

**点击(此处)折叠或打开**

1. /\*\*
2. \*    vmalloc - allocate virtually contiguous memory
3. \*    @size:        allocation size
4. \*    Allocate enough pages to cover @size from the page level
5. \*    allocator and map them into contiguous kernel virtual space.
6. \*
7. \*    For tight control over page level allocator and protection flags
8. \*    use \_\_vmalloc() instead.
9. \*/
10. void \*vmalloc(unsigned long size)
11. {
12. return \_\_vmalloc\_node\_flags(size, -1, GFP\_KERNEL | \_\_GFP\_HIGHMEM);
13. }

这里我们只用关注size即可，而vmalloc优先从高端内存分配，并且可以睡眠.  
继续：

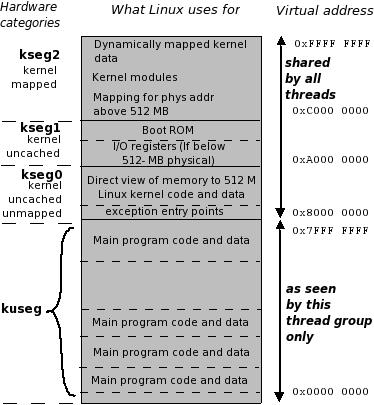
**点击(此处)折叠或打开**

1. static inline void \*\_\_vmalloc\_node\_flags(unsigned long size,
2. int node, gfp\_t flags)
3. {
4. return \_\_vmalloc\_node(size, 1, flags, PAGE\_KERNEL,
5. node, \_\_builtin\_return\_address(0));
6. }

重点看一下\_\_vmalloc\_node:

**点击(此处)折叠或打开**

1. /\*\*
2. \*    \_\_vmalloc\_node - allocate virtually contiguous memory
3. \*    @size:        allocation size
4. \*    @align:        desired alignment
5. \*    @gfp\_mask:    flags for the page level allocator
6. \*    @prot:        protection mask for the allocated pages
7. \*    @node:        node to use for allocation or -1
8. \*    @caller:    caller's return address
9. \*
10. \*    Allocate enough pages to cover @size from the page level
11. \*    allocator with @gfp\_mask flags. Map them into contiguous
12. \*    kernel virtual space, using a pagetable protection of @prot.
13. \*/
14. static void \*\_\_vmalloc\_node(unsigned long size, unsigned long align,
15. gfp\_t gfp\_mask, pgprot\_t prot,
16. int node, const void \*caller)
17. {
18. return \_\_vmalloc\_node\_range(size, align, VMALLOC\_START, VMALLOC\_END,
19. gfp\_mask, prot, node, caller);
20. }

因为这里提到了VMALLOC\_START和ＶＭＡＬＬＯＣ＿ＥＮＤ它们究竟是什么值呢？  
这里看了arm32和mips32的（根据架构虚拟地址分配不同而不同，比如mips就比较特殊）：  
在arch/mips/include/asm/pgtable-32.h中  
首先看mips虚拟地址分布图：  
  
从这个图里我们知道用户空间为2G（0x0-0x7fff ffff）,dma或者normal内存映射在kseg0（512M）/kseg1,而对于vmalloc申请的虚拟地址在kseg2中，当然还有其他一些特殊的映射比如io等.

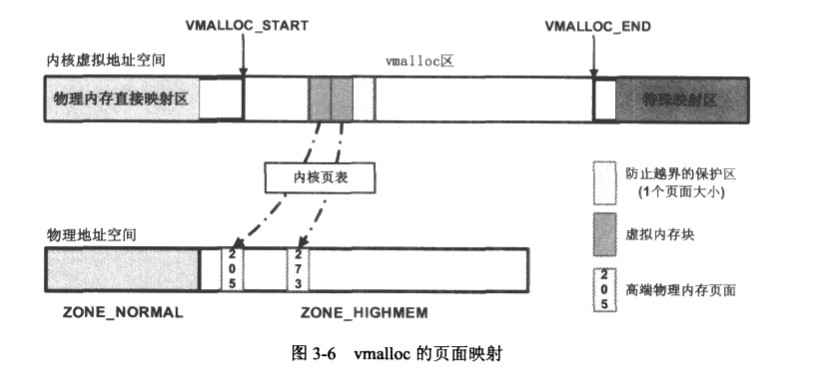
**点击(此处)折叠或打开**

1. #define VMALLOC\_START MAP\_BASE
2. #define PKMAP\_BASE        (0xfe000000UL)
3. #ifdef CONFIG\_HIGHMEM
4. # define VMALLOC\_END    (PKMAP\_BASE-2\*PAGE\_SIZE)
5. #else
6. # define VMALLOC\_END    (FIXADDR\_START-2\*PAGE\_SIZE)
7. #endif

在arch/arm/include/asm/pgtable.h

**点击(此处)折叠或打开**

1. /\*
2. \* Just any arbitrary offset to the start of the vmalloc VM area: the
3. \* current 8MB value just means that there will be a 8MB "hole" after the
4. \* physical memory until the kernel virtual memory starts. That means that
5. \* any out-of-bounds memory accesses will hopefully be caught.
6. \* The vmalloc() routines leaves a hole of 4kB between each vmalloced
7. \* area for the same reason. ;)
8. \*/
9. #define VMALLOC\_OFFSET        (8\*1024\*1024)
10. #define VMALLOC\_START        (((unsigned long)high\_memory + VMALLOC\_OFFSET) & ~(VMALLOC\_OFFSET-1))
11. #define VMALLOC\_END        0xff000000UL

在看一个图：  
  
我们知道物理内存简单分为三个区域：ZONE\_NORMAL、ZONE\_DMA、ZONE\_HIGHMEM  
vmalloc我们看到它是默认从ZONE\_HIGMEM里申请，但是这两个函数虚拟地址是保持一致的，即都占用了4G地址空间的内核虚拟地址.通过上面的图，**我们确定了虚拟地址从哪里分配，以及对于的物理空间从哪里分配。**  
下面看看 vmalloc核心实现：

**点击(此处)折叠或打开**

1. /\*\*
2. \*    \_\_vmalloc\_node\_range - allocate virtually contiguous memory
3. \*    @size:        allocation size
4. \*    @align:        desired alignment
5. \*    @start:        vm area range start
6. \*    @end:        vm area range end
7. \*    @gfp\_mask:    flags for the page level allocator
8. \*    @prot:        protection mask for the allocated pages
9. \*    @node:        node to use for allocation or -1
10. \*    @caller:    caller's return address
11. \*
12. \*    Allocate enough pages to cover @size from the page level
13. \*    allocator with @gfp\_mask flags. Map them into contiguous
14. \*    kernel virtual space, using a pagetable protection of @prot.
15. \*/
16. void \***\_\_vmalloc\_node\_range**(unsigned long size, unsigned long align,
17. unsigned long start, unsigned long end, gfp\_t gfp\_mask,
18. pgprot\_t prot, int node, const void \*caller)
19. {
20. struct vm\_struct \*area;
21. void \*addr;
22. unsigned long real\_size = size;
23. size = PAGE\_ALIGN(size);
24. if (!size || (size >> PAGE\_SHIFT) > totalram\_pages)
25. goto fail;
26. area = \_\_get\_vm\_area\_node(size, align, VM\_ALLOC | VM\_UNLIST,    // 分配虚拟地址空间 把vm\_struct 和vm\_area(红黑树机制)关联起来.
27. start, end, node, gfp\_mask, caller);
28. if (!area)
29. goto fail;
30. addr = \_\_vmalloc\_area\_node(area, gfp\_mask, prot, node, caller);  //计算需要申请的页面，申请page，然后修改页表完成映射.
31. if (!addr)
32. return NULL;
33. /\*
34. \* In this function, newly allocated vm\_struct is not added
35. \* to vmlist at \_\_get\_vm\_area\_node(). so, it is added here.
36. \*/
37. insert\_vmalloc\_vmlist(area);     //把vm\_struct插入 全局vmlist链表
38. /\*
39. \* A ref\_count = 3 is needed because the vm\_struct and vmap\_area
40. \* structures allocated in the \_\_get\_vm\_area\_node() function contain
41. \* references to the virtual address of the vmalloc'ed block.
42. \*/
43. kmemleak\_alloc(addr, real\_size, 3, gfp\_mask);    //内存泄露追踪
44. return addr;
45. fail:
46. warn\_alloc\_failed(gfp\_mask, 0,
47. "vmalloc: allocation failure: %lu bytes\n",
48. real\_size);
49. return NULL;
50. }

它的基本实现思路很简单：  
1. 分配虚拟地址空间   
2.对虚拟地址空间进行页表映射  
  
需要熟知 下面两个结构体：  
struct vmap\_area

**点击(此处)折叠或打开**

1. struct vmap\_area {
2. unsigned long va\_start;
3. unsigned long va\_end;
4. unsigned long flags;
5. struct rb\_node rb\_node;        /\* address sorted rbtree \*/
6. struct list\_head list;        /\* address sorted list \*/
7. struct list\_head purge\_list;    /\* "lazy purge" list \*/
8. struct vm\_struct \*vm;
9. struct rcu\_head rcu\_head;
10. };

vm\_struct \*area ：

**点击(此处)折叠或打开**

1. struct vm\_struct {
2. struct vm\_struct    \*next;
3. void            \*addr;
4. unsigned long        size;
5. unsigned long        flags;
6. struct page        \*\*pages;
7. unsigned int        nr\_pages;
8. phys\_addr\_t        phys\_addr;
9. const void        \*caller;
10. };

这里在说明一下vmalloc\_init的初始化.

**点击(此处)折叠或打开**

1. /\*
2. \* Set up kernel memory allocators
3. \*/
4. static void \_\_init mm\_init(void)
5. {
6. /\*
7. \* page\_cgroup requires contiguous pages,
8. \* bigger than MAX\_ORDER unless SPARSEMEM.
9. \*/
10. page\_cgroup\_init\_flatmem();
11. mem\_init();
12. kmem\_cache\_init();
13. percpu\_init\_late();
14. pgtable\_cache\_init();
15. vmalloc\_init();
16. }

其实在讲slab机制的时候已经说过。

**点击(此处)折叠或打开**

1. void \_\_init vmalloc\_init(void)
2. {
3. struct vmap\_area \*va;
4. struct vm\_struct \*tmp;
5. int i;
6. for\_each\_possible\_cpu(i) {
7. struct vmap\_block\_queue \*vbq;
8. vbq = &per\_cpu(vmap\_block\_queue, i);
9. spin\_lock\_init(&vbq->lock);
10. INIT\_LIST\_HEAD(&vbq->free);
11. }
12. /\* Import existing vmlist entries. \*/
13. for (tmp = vmlist; tmp; tmp = tmp->next) {                     // 在系统启动或者初始化之初，vmlist为空.
14. va = kzalloc(sizeof(struct vmap\_area), GFP\_NOWAIT);
15. va->flags = VM\_VM\_AREA;
16. va->va\_start = (unsigned long)tmp->addr;
17. va->va\_end = va->va\_start + tmp->size;
18. va->vm = tmp;
19. \_\_insert\_vmap\_area(va);
20. }
21. vmap\_area\_pcpu\_hole = VMALLOC\_END;
22. vmap\_initialized = true;
23. }

下面就说说\_\_get\_vm\_area\_node函数：

**点击(此处)折叠或打开**

1. static struct vm\_struct \*\_\_get\_vm\_area\_node(unsigned long size,
2. unsigned long align, unsigned long flags, unsigned long start,
3. unsigned long end, int node, gfp\_t gfp\_mask, const void \*caller)
4. {
5. struct vmap\_area \*va;
6. struct vm\_struct \*area;
7. BUG\_ON(in\_interrupt());
8. if (flags & VM\_IOREMAP) { // ioremap标志，映射的是设备内存
9. int bit = fls(size);
10. if (bit > IOREMAP\_MAX\_ORDER)
11. bit = IOREMAP\_MAX\_ORDER;
12. else if (bit < PAGE\_SHIFT)
13. bit = PAGE\_SHIFT;
14. align = 1ul << bit;
15. }
16. size = PAGE\_ALIGN(size);
17. if (unlikely(!size))
18. return NULL;
19. area = kzalloc\_node(sizeof(\*area), gfp\_mask & GFP\_RECLAIM\_MASK, node);
20. if (unlikely(!area))
21. return NULL;
22. /\*
23. \* We always allocate a guard page.
24. \*/
25. size += PAGE\_SIZE; // 多偏移一页，为了防止访问越界，由于多出来的一页并不映射，所以当访问的时候，会引发保护异常.
26. va = alloc\_vmap\_area(size, align, start, end, node, gfp\_mask);        // 申请vm\_area虚拟地址空间
27. if (IS\_ERR(va)) {
28. kfree(area);
29. return NULL;
30. }
31. /\*
32. \* When this function is called from \_\_vmalloc\_node\_range,
33. \* we do not add vm\_struct to vmlist here to avoid
34. \* accessing uninitialized members of vm\_struct such as
35. \* pages and nr\_pages fields. They will be set later.
36. \* To distinguish it from others, we use a VM\_UNLIST flag.
37. \*/
38. if (flags & VM\_UNLIST)   // 必然走这里
39. setup\_vmalloc\_vm(area, va, flags, caller);  // 关联vm\_struct 和 vm\_area
40. else
41. insert\_vmalloc\_vm(area, va, flags, caller);
42. return area;
43. }

这个函数核心就是alloc\_vmap\_area，这个很有趣的，之前我们讲到了vmalloc申请的虚拟地址范围，而它只传递了size而已，对于mips，x86，arm会有不同的虚拟空间.

**点击(此处)折叠或打开**

1. /\*
2. \* Allocate a region of KVA of the specified size and alignment, within the
3. \* vstart and vend.
4. \*/
5. static struct vmap\_area \*alloc\_vmap\_area(unsigned long size,
6. unsigned long align,
7. unsigned long vstart, unsigned long vend,
8. int node, gfp\_t gfp\_mask)
9. {
10. struct vmap\_area \*va;
11. struct rb\_node \*n;
12. unsigned long addr;
13. int purged = 0;
14. struct vmap\_area \*first;
15. BUG\_ON(!size);
16. BUG\_ON(size & ~PAGE\_MASK);
17. BUG\_ON(!is\_power\_of\_2(align));
18. va = kmalloc\_node(sizeof(struct vmap\_area),
19. gfp\_mask & GFP\_RECLAIM\_MASK, node);
20. if (unlikely(!va))
21. return ERR\_PTR(-ENOMEM);
22. retry:
23. spin\_lock(&vmap\_area\_lock);
24. /\*
25. \* Invalidate cache if we have more permissive parameters.
26. \* cached\_hole\_size notes the largest hole noticed \_below\_
27. \* the vmap\_area cached in free\_vmap\_cache: if size fits
28. \* into that hole, we want to scan from vstart to reuse
29. \* the hole instead of allocating above free\_vmap\_cache.
30. \* Note that \_\_free\_vmap\_area may update free\_vmap\_cache
31. \* without updating cached\_hole\_size or cached\_align.
32. \*/
33. if (!free\_vmap\_cache ||                              //第一次调用的时候 free\_vmap\_cache为空，后来即后边的代码line 105 ： free\_vmap\_cache = &va->rb\_node; 一般不为空 ；一般会发                                                           // 生align < cached\_align的情况,即会清除free\_vmap\_cache。有时候align比较大的时候，它会跳过一段虚拟地址空间.后面的申请由于没                                                            //有free\_vmap\_cache，所以它需要重新查询
34. size < cached\_hole\_size ||
35. vstart < cached\_vstart ||
36. align < cached\_align) {
37. nocache:
38. cached\_hole\_size = 0;
39. free\_vmap\_cache = NULL;
40. }
41. /\* record if we encounter less permissive parameters \*/
42. cached\_vstart = vstart;
43. cached\_align = align;
44. /\* find starting point for our search \*/
45. if (free\_vmap\_cache) {                                                  // 第一次使用的时候为空；当不为空时，它保持上次申请的节点，并初始化addr为va\_end.
46. first = rb\_entry(free\_vmap\_cache, struct vmap\_area, rb\_node);
47. addr = ALIGN(first->va\_end, align);
48. if (addr < vstart)
49. goto nocache;
50. if (addr + size - 1 < addr)
51. goto overflow;
52. } else {
53. addr = ALIGN(vstart, align);
54. if (addr + size - 1 < addr)
55. goto overflow;
56. n = vmap\_area\_root.rb\_node;                               // 同样vmap\_area\_root.rb\_node; 初始化也为空，第一次使用为空
57. first = NULL;
58. while (n) {                                               // 当不是第一申请，并且free\_cache为空的时候， 需要重新找到根节点即va\_start <= addr
59. struct vmap\_area \*tmp;
60. tmp = rb\_entry(n, struct vmap\_area, rb\_node);
62. if (tmp->va\_end >= addr) {
63. first = tmp;
64. if (tmp->va\_start <= addr)
65. break;
66. n = n->rb\_left;
67. } else
68. n = n->rb\_right;
69. }
70. if (!first)
71. goto found;
72. }
73. /\* from the starting point, walk areas until a suitable hole is found \*/
74. while (addr + size > first->va\_start && addr + size <= vend) {                // 当不是第一申请，并且free\_cache为空的时候,查询红黑树节点，找到合适的空间地址.
75. if (addr + cached\_hole\_size < first->va\_start)
76. cached\_hole\_size = first->va\_start - addr;
77. addr = ALIGN(first->va\_end, align);
78. if (addr + size - 1 < addr)
79. goto overflow;
81. if (list\_is\_last(&first->list, &vmap\_area\_list))     // 默认不会在这里操作。也就是说它没有元素.
82. goto found;
83. first = list\_entry(first->list.next,
84. struct vmap\_area, list);
85. }
86. found:
87. if (addr + size > vend)
88. goto overflow;
89. va->va\_start = addr;
90. va->va\_end = addr + size;
91. va->flags = 0;
92. \_\_insert\_vmap\_area(va);                           // 添加到红黑树 vmap\_area\_root
93. free\_vmap\_cache = &va->rb\_node;                  // 初始化free\_vmap\_cache ，它会影响后续虚拟空间的申请.
94. spin\_unlock(&vmap\_area\_lock);
95. BUG\_ON(va->va\_start & (align-1));
96. BUG\_ON(va->va\_start < vstart);
97. BUG\_ON(va->va\_end > vend);
98. return va;
99. overflow:
100. spin\_unlock(&vmap\_area\_lock);
101. if (!purged) {
102. purge\_vmap\_area\_lazy();
103. purged = 1;
104. goto retry;
105. }
106. if (printk\_ratelimit())
107. printk(KERN\_WARNING
108. "vmap allocation for size %lu failed: "
109. "use vmalloc= to increase size.\n", size);
110. kfree(va);
111. return ERR\_PTR(-EBUSY);
112. }

既然我们已经开辟了虚拟地址空间，那么还需要做的当然是和页面一一映射起来.  
看函数\_\_vmalloc\_area\_node：

**点击(此处)折叠或打开**

1. static void \*\_\_vmalloc\_area\_node(struct vm\_struct \*area, gfp\_t gfp\_mask,
2. pgprot\_t prot, int node, const void \*caller)
3. {
4. const int order = 0;
5. struct page \*\*pages;
6. unsigned int nr\_pages, array\_size, i;
7. gfp\_t nested\_gfp = (gfp\_mask & GFP\_RECLAIM\_MASK) | \_\_GFP\_ZERO;
8. nr\_pages = (area->size - PAGE\_SIZE) >> PAGE\_SHIFT; //申请多少pages
9. array\_size = (nr\_pages \* sizeof(struct page \*));   //需要多大的存放page指针的空间 .
10. area->nr\_pages = nr\_pages;
11. /\* Please note that the recursion is strictly bounded. \*/
12. if (array\_size > PAGE\_SIZE) {                          // 这里默认page\_size 为4k 即4096 ，地址32位的话，相当于申请1024个pages：4M空间
13. pages = \_\_vmalloc\_node(array\_size, 1, nested\_gfp|\_\_GFP\_HIGHMEM,
14. PAGE\_KERNEL, node, caller);
15. area->flags |= VM\_VPAGES;
16. } else {
17. pages = kmalloc\_node(array\_size, nested\_gfp, node);    // 小于一页，则直接利用slab机制申请物理空间地址 给pages.
18. }
19. area->pages = pages;
20. area->caller = caller;
21. if (!area->pages) {
22. remove\_vm\_area(area->addr);
23. kfree(area);
24. return NULL;
25. }
26. for (i = 0; i < area->nr\_pages; i++) {              //  每次申请一个page利用alloc\_page直接申请物理页面
27. struct page \*page;
28. gfp\_t tmp\_mask = gfp\_mask | \_\_GFP\_NOWARN;
29. if (node < 0)
30. page = alloc\_page(tmp\_mask);
31. else
32. page = alloc\_pages\_node(node, tmp\_mask, order);
33. if (unlikely(!page)) {
34. /\* Successfully allocated i pages, free them in \_\_vunmap() \*/
35. area->nr\_pages = i;
36. goto fail;
37. }
38. area->pages[i] = page;             // 分配的地址存放在指针数组.
39. }
40. if (map\_vm\_area(area, prot, &pages)) // 修改页表 ,一页一页的实现映射，以及flush cache保持数据的一致性；对页面映射和操作感兴趣的可以深入看看这个函数.
41. goto fail;
42. return area->addr;
43. fail:
44. warn\_alloc\_failed(gfp\_mask, order,
45. "vmalloc: allocation failure, allocated %ld of %ld bytes\n",
46. (area->nr\_pages\*PAGE\_SIZE), area->size);
47. vfree(area->addr);
48. return NULL;
49. }

而insert\_vmalloc\_vmlist很明显把vm\_struct插入到vmlist。  
那么就完成了整个过程，没有想象的复杂，当然对内存有了更多的认识，这里还需要说一下，一般情况下有高端内存会比没有的好些，防止了vmalloc申请的时候造成的TLB抖动等问题，更少的破坏normal空间。  
可以通过proc来查看vmalloc的一下信息：

**点击(此处)折叠或打开**

1. cat /proc/vmallocinfo
2. 0xc0002000-0xc0045000 274432 jffs2\_zlib\_init+0x24/0xa4 pages=66 vmalloc
3. 0xc0045000-0xc0051000 49152 jffs2\_zlib\_init+0x40/0xa4 pages=11 vmalloc
4. 0xc0051000-0xc0053000 8192 brcmnand\_create\_cet+0x244/0x788 pages=1 vmalloc
5. 0xc0053000-0xc0055000 8192 ebt\_register\_table+0x98/0x39c pages=1 vmalloc

还有：

**点击(此处)折叠或打开**

1. # cat /proc/vmstat
2. #cat /proc/meminfo

Android ion:

*ION 和 VIDEOBUF2、DMA-BUF、V4L2 等结合 使用分析*

# 内存管理 —— ION

18 November 2017

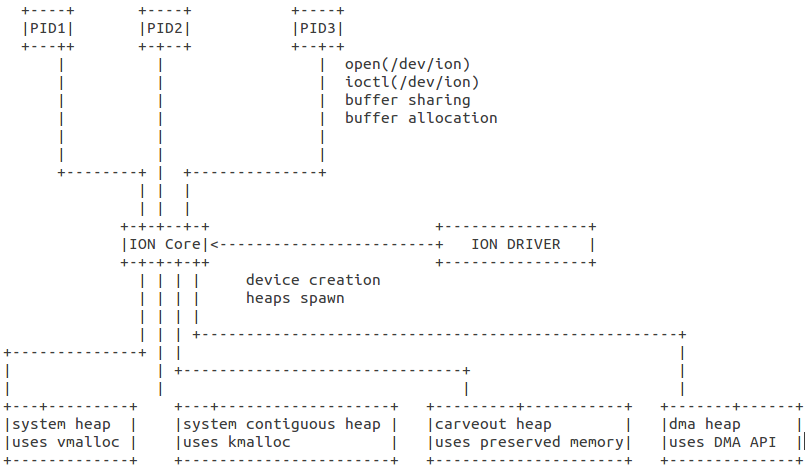
ION 是当前 Android 流行的内存分配管理机制，在多媒体部分中使用的最多，例如从 Camera 到 Display，从 Mediaserver 到 Surfaceflinger，都会利用 ION 进行内存分配管理。 ION 的前任是 PMEM，关于 PMEM 我在 M030/M04X 项目中有接触过，后来由于 PMEM 的一些局限性，Google 推出了 ION 来取代 PMEM，当前 ION 已经融合到 Linux 主线，被广泛使用。 对于魅族，从 M65 项目开始，相机的内存分配管理就已经利用 ION 进行了，本文会结合 M65、M76 和 M86 项目开发介绍下我对 ION 内存管理机制的理解和感悟。

## ION 基本概念

ION，最显著的特点是它可以被用户空间的进程之间或者内核空间的模块之间进行内存共享，而且这种共享可以是零拷贝的。在实际使用中，ION 和 VIDEOBUF2、DMA-BUF、V4L2 等结合的很紧密。本文主要介绍 ION，其它子系统感兴趣的话后续会陆续进行介绍。

ION 是在各种 heaps 上分配内存，通过 ion\_buffer 来描述所分配的内存。

下图展示了 ION 的基本框架。图中 PID1、PID2、PID3 表示用户空间进程。ION core 表示 ION 核心层，它提供设备创建、注册等服务，同时提供统一的接口给用户使用。ION Driver 利用 ION core 对相应功能进行实现，可以说它是具体平台相关的，例如 SAMSUNG 平台、QUALCOMM 平台和 MTK 平台都会依据自己的特性开发相应的 ION Driver。

[](https://kernel.meizu.com/images/posts/2017/11/ION_ARCHI.png)ION\_ARCHI

上图虽然描述的是用户空间进程使用 ION 的情形，但实际上，在内核空间同样可以直接使用 ION 来分配、管理内存。例如 M76、M86 平台的相机驱动，都有直接使用 ION 分配和管理内存。

## 主要数据结构

数据结构是程序设计的基础，代码看的多了，其实可以从数据结构看出其能提供的基本功能和大致用法。 为了抓住纲领，本文将抓住 ION 的主要数据结构进行介绍。

### ion\_device

ion\_device 是 ION 很重要很基础的数据结构，用 struct ion\_device 结构体描述，一般在一个系统中只有一个本实例。例如在 M86 中，就是在 exynos\_ion\_v2.c 的 exynos\_ion\_probe() 函数中创建了系统唯一的本实例。

**struct** ion\_device {

**struct** miscdevice dev;

**struct** rb\_root buffers;

**struct** mutex buffer\_lock;

**struct** rw\_semaphore lock;

**struct** plist\_head heaps;

long (\*custom\_ioctl)(**struct** ion\_client \*client, unsigned int cmd,

unsigned long arg);

**struct** rb\_root clients;

**struct** dentry \*debug\_root;

**struct** dentry \*heaps\_debug\_root;

**struct** dentry \*clients\_debug\_root;

**struct** semaphore vm\_sem;

atomic\_t page\_idx;

**struct** vm\_struct \*reserved\_vm\_area;

pte\_t \*\*pte;

...};

struct ion\_device 其实是 ION 的核心结构，不过由于对于使用 ION 的用户而言是屏蔽的，即如果是单纯使用 ION，不需要直接和本结构打交道。但是想完全的理解 ION，需要对其有所了解。 struct ion\_device 是由 ion\_device\_create() [ion.c] 分配、初始化。

* dev 成员，是 struct miscdevice 类型，所以可想而知 ION 是作为 MISC 设备注册进系统的。从这点还可以看出来，用户空间使用 ION 时必定需要使用 open 啊，ioctl 啊系统调用。事实也正是如此。
* heaps 成员，在 M86 使用过的 KERNEL LINUX 3.10 中是 struct plist\_head 类型，但是在此之前并不是此类型，例如在 M65 使用过的 KERNEL LINUX 3.4 中是 struct rb\_root 类型。可见随着 KERNEL 和 ION 的演进，struct ion\_device 的实现会有所改变。本字段管理的是属于本 struct ion\_device 的所有 struct ion\_heap 实例。
* clients 成员，是 struct rb\_root 类型，struct rb\_root 是红黑树，属于二叉树的一种。本字段管理的是 struct ion\_client 实例。

### ion\_client

struct ion\_client 是由 ion\_client\_create() [ion.c] 创建，在创建时必须指定上文提到的 struct ion\_device 实例。

**struct** ion\_client {

**struct** rb\_node node;

**struct** ion\_device \*dev;

**struct** rb\_root handles;

**struct** idr idr;

**struct** mutex lock;

**const** char \*name;

char \*display\_name;

int display\_serial;

**struct** task\_struct \*task;

pid\_t pid;

**struct** dentry \*debug\_root;};

* node 成员，是 struct rb\_node 结构类型，用于将本 struct ion\_client 实例加入到 struct ion\_device，具体的是 struct ion\_device::clients。
* device 成员，是 struct ion\_device 指针结构类型，指向所属的 struct ion\_device 实例。
* handles 成员，是 struct rb\_root 结构类型，管理其所拥有的 handle，即 struct ion\_handle 实例。一个 struct ion\_handle 实例表示一个 buffer，即 struct ion\_buffer 实例。而 struct ion\_buffer 就是从 heap，即 struct ion\_heap 中分配的内存。

### ion\_heap

struct ion\_heap 表示 ION 中的重要概念 heap。系统会通过链表或者红黑树，这取决与你所使用的 KERNEL 版本，来管理所有的 heap，这些 heap 可用 struct ion\_device::heaps 字段来寻找。

**struct** ion\_heap {

**struct** plist\_node node;

**struct** ion\_device \*dev;

**enum** ion\_heap\_type type;

**struct** ion\_heap\_ops \*ops;

unsigned long flags;

unsigned int id;

**const** char \*name;

**struct** shrinker shrinker;

**struct** list\_head free\_list;

size\_t free\_list\_size;

spinlock\_t free\_lock;

wait\_queue\_head\_t waitqueue;

**struct** task\_struct \*task;

int (\*debug\_show)(**struct** ion\_heap \*heap, **struct** seq\_file \*, void \*);};

* node 成员，是 struct plist\_node 结构，用于将本 heap 实例加入到 struct ion\_device 所管理的链表中，详情可以参见 ion\_device\_add\_heap() [ion.c] 函数。
* dev 成员，是 struct ion\_device 结构体指针类型，用于指示本 heap 挂在哪一个 struct ion\_device 实例下了。
* type 成员，是 enum ion\_heap\_type 类型，用于表示本 heap 属于哪种类型。用户在使用 ION 分配内存时需要指定 heap 的种类。关于本字段，后续会结合使用方法进行更详细的介绍。
* ops 成员，是 struct ion\_heap\_ops 类型，它很重要！本字段提供的回调函数是用于从本 heap 中分配内存时所时用的。请参见 ion\_buffer\_create() [ion.c]，它会调用 struct ion\_heap\_ops::allocate() 等回调函数。

**struct** ion\_heap\_ops {

int (\*allocate)(**struct** ion\_heap \*heap,

**struct** ion\_buffer \*buffer, unsigned long len,

unsigned long align, unsigned long flags);

void (\*free)(**struct** ion\_buffer \*buffer);

int (\*phys)(**struct** ion\_heap \*heap, **struct** ion\_buffer \*buffer,

ion\_phys\_addr\_t \*addr, size\_t \*len);

**struct** sg\_table \* (\*map\_dma)(**struct** ion\_heap \*heap,

**struct** ion\_buffer \*buffer);

void (\*unmap\_dma)(**struct** ion\_heap \*heap, **struct** ion\_buffer \*buffer);

void \* (\*map\_kernel)(**struct** ion\_heap \*heap, **struct** ion\_buffer \*buffer);

void (\*unmap\_kernel)(**struct** ion\_heap \*heap, **struct** ion\_buffer \*buffer);

int (\*map\_user)(**struct** ion\_heap \*mapper, **struct** ion\_buffer \*buffer,

**struct** vm\_area\_struct \*vma);

int (\*shrink)(**struct** ion\_heap \*heap, gfp\_t gfp\_mask, int nr\_to\_scan);

void (\*preload) (**struct** ion\_heap \*heap, unsigned int count,

unsigned int flags, **struct** ion\_preload\_object obj[]);

};

* id 成员，也很重要，它可表示优先级，在分配内存时选择哪一个 heap 有关，必须唯一。

### ion\_handle

struct ion\_handle 其实就是表示 buffer，用户空间常用它来表示 buffer。本结构通过 ion\_handle\_create() [ion.c] 分配、初始化。

**struct** ion\_handle {

**struct** kref ref;

**struct** ion\_client \*client;

**struct** ion\_buffer \*buffer;

**struct** rb\_node node;

unsigned int kmap\_cnt;

int id;};

* ref 成员，是 struct kref 结构类型，它在内核中被广泛的用来表示引用计数。ion\_handle 的创建和销毁都与其有关，下文会有介绍。
* client 成员，是 struct ion\_client 指针类型，指向其所述的 ion\_client 实例。
* buffer 成员，是 struct ion\_buffer 指针类型，指向真正的 buffer 所在，它可以说是 stuct ion\_handle 的核心成员了。

下面就来介绍 struct ion\_buffer 结构。在分配内存时，也是先通过 ion\_buffer\_create() 创建 ion\_buffer 实例，然后交给 ion\_handle\_create() 创建 ion\_handle 实例。

### ion\_buffer

struct ion\_buffer 很重要，通过 ION 分配的内存就是通过它表示的。它和上面提到的 ion\_handle 的区别主要在于一个是用户空间使用的，一个是内核空间使用的。即虽然常用的接口函数中使用的是 struct ion\_handle，但实际上真正表示内存的其实是 struct ion\_buffer。

**struct** ion\_buffer {

**struct** kref ref;

...

**struct** ion\_device \*dev;

**struct** ion\_heap \*heap;

unsigned long flags;

unsigned long private\_flags;

size\_t size;

**union** {

void \*priv\_virt;

ion\_phys\_addr\_t priv\_phys;

};

**struct** mutex lock;

int kmap\_cnt;

void \*vaddr;

int dmap\_cnt;

**struct** sg\_table \*sg\_table;

**struct** page \*\*pages;

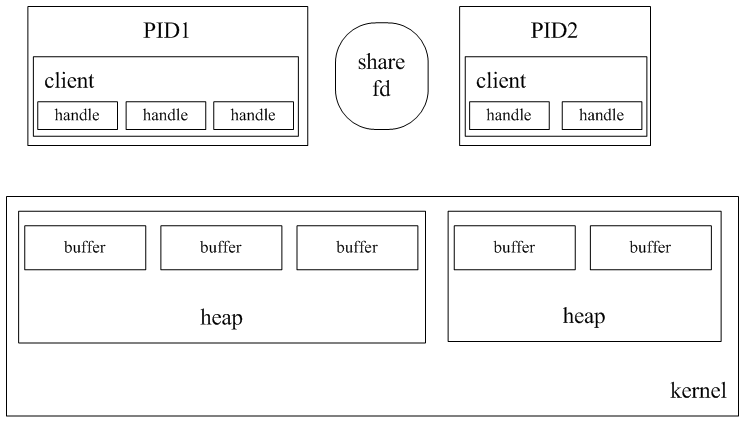
**struct** list\_head vmas;

**struct** list\_head iovas;

...};

* ref 成员，是 struct kref 结构实例，维护了本 ion\_buffer 的引用计数。当引用计数为 0 时会释放该 buffer，即 struct ion\_heap\_ops::free 会被调用。分配用 ION\_IOC\_ALLOC 型 ioctl 系统调用，相应的释放用 ION\_IOC\_FREE 型 ioctl 系统调用。
* size 成员，当然是本 buffer 所表示的空间的大小，用字节表示。
* priv\_virt 成员，是所分配内存的虚拟地址啦，它常与 struct sg\_table，或者封装它的结构，有关。它不是我们在内核中读写时所需的内核虚拟地址啦，内核虚拟地址使用 vaddr 成员来表示的。一般而言，物理内存不连续的，使用本字段；否则使用下面的 priv\_phys 字段，如 struct ion\_heap\_ops contig\_heap\_ops。
* priv\_phys 成员，表示所分配的内存的物理地址。它适用于分配的物理内存是连续的 ion heap。这种连续的物理内存：在将其映射到用户空间时，即获取用户空间虚拟地址，可以使用 remap\_pfn\_range() [memory.c] 这个方便的接口；在将其映射到内核空间时，即获取内核虚拟地址，可以使用 vmap() [vmalloc.c] 这个方便的接口。例子详见 struct ion\_heap\_ops contig\_heap\_ops [exynos\_ion.c]。priv\_virt 成员和 priv\_phys 成员组成了一个联合体，其实都表示地址，只不过不同的场景下具体用的不一样而已。
* kmap\_cnt 成员，记录本 buffer 被映射到内核空间的次数。
* vaddr 成员，是本 buffer 对应的内核虚拟地址。当 kmap\_cnt 不为 0 时有效。可以通过 ion\_map\_kernel() [ion.c] 来获取本 buffer 对应的内核虚拟地址。ion\_map\_kernel() [ion.c] 实际上调用的是相应 struct ion\_heap\_ops::map\_kernel 回调函数获取相应的虚拟地址的。
* dmap\_cnt 成员，记录本 buffer 被 mapped for DMA 的次数。
* sg\_table 成员，是 struct sg\_table 结构体类型的指针。本字段与 DMA 操作有关，而且仅仅在 dmap\_cnt 成员变量不为 0 时是有效的。可以通过 ion\_buffer\_create() [ion.c] 来初始化本成员变量，该函数实际上是调用相应 ion\_heap 所属的 struct ion\_heap\_ops::map\_dma 回调函数获取本字段的值的。
* dirty 成员，表示 bitmask。即以位图表示本 buffer 的哪一个 page 是 dirty 的，即不能直接用于 DMA。dirty 表示 DMA 的不一致性，即 CPU 缓存中的内容与内存中的实际内容不一样。

事实上，ION 涉及到的数据结构还有很多，这里列举的都是一些非常重要的。 下图展示了上文介绍到的数据结构的基本关系。

[](https://kernel.meizu.com/images/posts/2017/11/ION_RELAT.png)ION\_RELAT

## 重要函数分析

函数对数据进行处理，完成特定的任务，体现算法的具体实现。

### ion\_device\_create

前面分析 ION 的一些核心数据结构时曾经指出，ION 会注册进 MISC 设备，这样用户空间就可以像使用 MISC 设备一样使用 ION 进行内存分配了。 先来看 ION 是如何注册进 MISC 子系统的。

**struct** ion\_device \*ion\_device\_create(long (\*custom\_ioctl)

(**struct** ion\_client \*client,

unsigned int cmd,

unsigned long arg)){

**struct** ion\_device \*idev;

int ret;

*/\**

*\* 分配 struct ion\_device 实例*

*\*/*

idev = kzalloc(**sizeof**(**struct** ion\_device), GFP\_KERNEL);

...

*/\**

*\* 如前面所说，dev 成员是 struct miscdevice 结构体类型，*

*\* 这里初始化其相关字段，名字指定为 ”ion”，所以提供*

*\* 给用户空间调用的设备节点名为 /dev/ion。*

*\* 用户空间操作该设备节点时，ION 驱动中响应的函数集位于*

*\* ion\_fops*

*\*/*

idev->dev.minor = MISC\_DYNAMIC\_MINOR;

idev->dev.name = "ion";

idev->dev.fops = &ion\_fops;

idev->dev.parent = NULL;

ret = misc\_register(&idev->dev);

...

*/\**

*\* 注册调试信息接口*

*\*/*

idev->debug\_root = debugfs\_create\_dir("ion", NULL);

...

idev->heaps\_debug\_root = debugfs\_create\_dir("heaps", idev->debug\_root);

...

idev->clients\_debug\_root = debugfs\_create\_dir("clients",

idev->debug\_root);

...

debugfs\_done:

idev->custom\_ioctl = custom\_ioctl;

idev->buffers = RB\_ROOT;

mutex\_init(&idev->buffer\_lock);

init\_rwsem(&idev->lock);

plist\_head\_init(&idev->heaps);

idev->clients = RB\_ROOT;

...

*/\* backup of ion device: assumes there is only one ion device \*/*

g\_idev = idev;

**return** idev;}

本函数最重要的是分配并初始化了核心 struct ion\_device 实例，并将其和 MISC 设备结合起来，这样用户空间就可以通过 open()、ioctl() 等系统调用使用它了。

### ion\_open

用户空间要想使用 ION 进行内存分配，首先必须对设备节点 /dev/ion 进行 open() 系统调用。

**static** int ion\_open(**struct** inode \*inode, **struct** file \*file){

**struct** miscdevice \*miscdev = file->private\_data;

**struct** ion\_device \*dev = container\_of(miscdev, **struct** ion\_device, dev);

**struct** ion\_client \*client;

char debug\_name[64];

…

snprintf(debug\_name, 64, "%u", task\_pid\_nr(current->group\_leader));

*/\**

*\* 创建struct ion\_client 实例*

*\*/*

client = ion\_client\_create(dev, debug\_name);

...

file->private\_data = client;

**return** 0;}

ion\_open() 函数最重要的作用就是创建了 struct ion\_client 实例。这样，后续就可以利用 ioctl 系统调用从其中分配内存了。

ION 系统提供的 ioctl 类型有很多，常用的有 ION\_IOC\_ALLOC、ION\_IOC\_FREE、ION\_IOC\_SHARE 和 ION\_IOC\_IMPORT 等等。

#define ION\_IOC\_ALLOC \_IOWR(ION\_IOC\_MAGIC, 0, struct ion\_allocation\_data)

#define ION\_IOC\_FREE \_IOWR(ION\_IOC\_MAGIC, 1, struct ion\_handle\_data)

#define ION\_IOC\_SHARE \_IOWR(ION\_IOC\_MAGIC, 4, struct ion\_fd\_data)

下面就抽出几个典型的进行分析。

### ion\_alloc

这是当用户空间执行 ION\_IOC\_ALLOC 型 ioctl() 系统调用时所执行的。

**struct** ion\_handle \*ion\_alloc(**struct** ion\_client \*client, size\_t len,

size\_t align, unsigned int heap\_id\_mask,

unsigned int flags){

**struct** ion\_handle \*handle;

**struct** ion\_device \*dev = client->dev;

**struct** ion\_buffer \*buffer = NULL;

**struct** ion\_heap \*heap;

int ret;

...

*/\**

*\* len 是用户空间想分配的内存大小，驱动中会将其进行页对齐*

*\*/*

len = PAGE\_ALIGN(len);

...

down\_read(&dev->lock);

*/\**

*\* 用户空间会指定其想从哪种 heap 分配内存，ION 驱动*

*\* 会对其进行检查并找到最合适的。*

*\*/*

heap\_id\_mask = ion\_parse\_heap\_id(heap\_id\_mask, flags);

*/\**

*\* 从系统所有的 heap 中找到最合适的 heap 并分配内存，*

*\* 创建 struct ion\_buffer 实例。*

*\*/*

plist\_for\_each\_entry(heap, &dev->heaps, node) {

*/\* if the caller didn't specify this heap id \*/*

**if** (!((1 << heap->id) & heap\_id\_mask))

**continue**;

buffer = ion\_buffer\_create(heap, dev, len, align, flags);

**if** (!IS\_ERR(buffer))

**break**;

}

up\_read(&dev->lock);

...

*/\**

*\* 在对创建的 ion\_buffer 实例进行了一系列的 sanity 检查后*

*\* 利用其创建 struct ion\_handle 实例。*

*\*/*

handle = ion\_handle\_create(client, buffer);

...

mutex\_lock(&client->lock);

*/\**

*\* 将 struct ion\_handle 实例加入到其所属的 struct ion\_client*

*\*/*

ret = ion\_handle\_add(client, handle);

mutex\_unlock(&client->lock);

...

**return** handle;}

### ion\_free

这是当用户空间执行 ION\_IOC\_FREE 型 ioctl() 系统调用时所执行的。

void ion\_free(**struct** ion\_client \*client, **struct** ion\_handle \*handle){

bool valid\_handle;

BUG\_ON(client != handle->client);

mutex\_lock(&client->lock);

valid\_handle = ion\_handle\_validate(client, handle);

**if** (!valid\_handle) {

WARN(1, "%s: invalid handle passed to free.**\n**", \_\_func\_\_);

mutex\_unlock(&client->lock);

**return**;

}

mutex\_unlock(&client->lock);

ion\_handle\_put(client, handle);}

此函数比较简单，重点就是通过 ion\_handle\_put() 来对上文提到的 struct ion\_handle::ref 这个 reference count 减一，当 ref 减到 0 时，就会调用 ion\_handle\_destroy() 来销毁 ion\_handle 实例。

从前文的分析可知，用户空间在利用 ION 分配内存时，需要指定具体的 heap mask，即告知 ION 想从哪种 heap 分配内存。 下面就来介绍下。

## HEAP 种类

以下是通过 ION 分配内存时，可能会使用到的 heap mask。

#define ION\_HEAP\_SYSTEM\_MASK (1 << 0)

#define ION\_HEAP\_SYSTEM\_CONTIG\_MASK (1 << 1)

#define ION\_HEAP\_EXYNOS\_CONTIG\_MASK (1 << 4)

#define ION\_HEAP\_EXYNOS\_MASK (1 << 5)

以 M65 项目为例，系统定义了 4 种 heap，见 dev-ion.c。

### ION\_HEAP\_TYPE\_SYSTEM

本 heap 的名字为：”ion\_noncontig\_heap”。相应的 heap mask 为 1，即 ION\_HEAP\_SYSTEM\_MASK。 在本 struct ion\_heap 上分配内存的操作集是： struct ion\_heap\_ops system\_heap\_ops [ion\_system\_heap.c] 其内存可以从 HIGHMEM 中分配，所以其物理内存可能不连续。 在调试 M65 CAMERA 的 HDR 功能时，一开始发现其写入速度很慢很慢，经过调查，后来在通过 ION 分配内存时指定 ION\_FLAG\_CACHED | ION\_FLAG\_CACHED\_NEEDS\_SYNC | ION\_FLAG\_PRESERVE\_KMAP 标志后速度得到明显提高。

### ION\_HEAP\_TYPE\_SYSTEM\_CONTIG

本 heap 的名字为：”ion\_contig\_heap”，相应的 heap mask 为 2，即 ION\_HEAP\_SYSTEM\_CONTIG\_MASK。 在本 struct ion\_heap 上分配内存的操作集是： struct ion\_heap\_ops kmalloc\_ops [ion\_system\_heap.c] 其内存分配 allocate 回调函数为：ion\_system\_contig\_heap\_allocate() [ion\_system\_heap.c]，该函数很简单，就是利用 kzalloc() 分配内存。

### ION\_HEAP\_TYPE\_EXYNOS

本 heap 的名字为：”exynos\_noncontig\_heap”，相应的 heap mask 为 32，即 ION\_HEAP\_EXYNOS\_MASK。 在本 struct ion\_heap 上分配内存的操作集是： struct ion\_heap\_ops vmheap\_ops [exynos\_ion.c] 其内存可以从 HIGHMEM 中分配，所以其物理内存不一定连续。详情可以见其 allocate 回调函数 ion\_exynos\_heap\_allocate() [exynos\_ion.c]。

### ION\_HEAP\_TYPE\_EXYNOS\_CONTIG

本 heap 的名字为：”exynos\_contig\_heap”。相应的 heap mask 为 16，即 ION\_HEAP\_EXYNOS\_CONTIG\_MASK。 在本 struct ion\_heap 上分配内存的操作集是： struct ion\_heap\_ops contig\_heap\_ops [exynos\_ion.c] 内存由 CMA 分配，所以可以保证其物理地址是连续的。详情可以参见其 allocate 回调函数 ion\_exynos\_contig\_heap\_allocate() [exynos\_ion.c]。 因为本函数是利用 CMA 分配内存，所以可以推测肯定有地方预留了 CMA 所需的物理内存。分析代码后可以发现，这个地方就位于 mach-m65.c 文件。 M65 的相机驱动使用了本 ION\_HEAP\_TYPE\_EXYNOS\_CONTIG 类型的 heap 来分配内存。 按道理，当用户空间获取了 struct ion\_handle 实例后，就已经完成了 ION 内存的分配任务。但实际上，为了在不同的进程间，甚至在用户空间和内核空间共享这段内存使用，用户空间还通常需要调用 ION\_IOC\_SHARE 型ioctl()，获取 ion buffer 相关的 fd，这就和 dma\_buf 子系统联系起来了。

int ion\_share\_dma\_buf\_fd(**struct** ion\_client \*client, **struct** ion\_handle \*handle){

**struct** dma\_buf \*dmabuf;

int fd;

*/\**

*\* 前面已经说过 struct ion\_handle 其实就是对 struct ion\_buffer*

*\* 的封装，这里利用 struct ion\_client 实例和 struct ion\_handle 实例*

*\* 创建了一个 dma\_buf。*

*\*/*

dmabuf = ion\_share\_dma\_buf(client, handle);

...

*/\**

*\* 重要，这样就将 struct ion\_buffer 对应的内存转化*

*\* 为文件描述符了。将文件描述符传递给其它进程或者*

*\* 传给内核空间，其它进程或者内核空间就可以使用其内存了*

*\*/*

fd = dma\_buf\_fd(dmabuf, O\_CLOEXEC);

...

**return** fd;}

到这里，ION 使用过程中涉及到的重要函数都已经介绍完全。当然要完全理解这些函数的细节，需要用户对 dma\_buffer 有一定的了解。对 dma\_buffer 的介绍不属于本文的范围，有兴趣的话可以参见我写的其它相关文档。

## 结语

关于 Android ION 内存管理机制的介绍就到这里。本文先介绍了什么是 ION，为什么要用 ION。ION 是为了解决内存碎片管理而引入的通用内存管理器，用于取代 PMEM 机制。然后介绍了下 ION 中的重要数据结构，对 struct ion\_device、struct ion\_client、struct ion\_heap、struct ion\_handle 和 struct ion\_buffer 进行了详细的介绍，并对它们之间的关系进行了阐述。接着，从使用 ION 的场景出发，介绍了一些重要的函数，例如 ion\_alloc()，并以实际的相机开发为例介绍了系统中各个 ion\_heap 的种类和各自的内存特性。本文还对进程之间、内核空间和用户空间之间的内存共享进行了介绍。

## 参考资料

* [The Android ION memory allocator](https://lwn.net/Articles/480055/" \t "https://kernel.meizu.com/_blank)
* [Integrating the ION memory allocator](https://lwn.net/Articles/565469/" \t "https://kernel.meizu.com/_blank)